Garbage Collection Programming Guide

General: Memory Management

2008-11-19

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures and Listings

Introduction to Garbage Collection

In Mac OS X version 10.5, the Cocoa programming environment is enhanced with automatic memory management—commonly known as "garbage collection." The "traditional" system of memory management (using retain, release, and autorelease pools)—herein referred to as "reference counted"—coexists both for binary compatibility with existing programs as well as for those that choose to not make use of the new facility. Garbage collection is hence an opt-in system.

These documents describe the complete garbage collection system provided for Cocoa, the functionality provided, and some of the issues that arise if you adopt this technology.

Who Should Read This Document?

If you are developing applications using Cocoa, you should read at least "Garbage Collection for Cocoa Essentials" (page 11) to gain an understanding of the garbage collection system. It is strongly recommended that you also read "Adopting Garbage Collection" (page 17) and "Implementing a finalize Method" (page 31). You are expected to already understand the Objective-C language (see *The Objective-C 2.0 Programming Language*) and to have some familiarity with Cocoa.

Organization of This Document

The following articles explain the problems the garbage collection system addresses, the solutions it provides, its basic functionality, and common tasks you might perform:

- "Garbage Collection for Cocoa Essentials" (page 11) describes the essential details of the garbage collection system for Cocoa. At a minimum, you should read this article.
- "Adopting Garbage Collection" (page 17) describes issues related to adopting garbage collection.
- "Architecture" (page 19) describes the design goals and architecture of the technology, and the benefits you get from using it.
- "Using Garbage Collection" (page 25) describes some of the features you can take advantage of when you use garbage collection, and some of subtleties you need to be aware of.
- "Implementing a finalize Method" (page 31) describes how to correctly implement a `finalize` method.
- "Inapplicable Patterns" (page 35) describes Cocoa programming patterns that are not applicable to garbage collection.
- "Using Core Foundation with Garbage Collection" (page 39) describes how to use Core Foundation objects with garbage collection.
- "Garbage Collection API" (page 45) provides a summary of API used in garbage collection.

See Also

The following documents provide information about related aspects of Cocoa and the Objective-C language.

- *The Objective-C 2.0 Programming Language* describes object-oriented programming and describes the Objective-C programming language.
- *Objective-C 2.0 Runtime Reference* describes the data structures and functions of the Objective-C runtime support library.
- *Memory Management Programming Guide for Cocoa* addresses Cocoa's object-ownership policy for manual memory management and related techniques for creating, copying, retaining, and disposing of objects.
- Garbage Collection Release Notesâ aLÂMOSÂ XÂ v10.5.0 provides information about the current release of the technology.

Garbage Collection for Cocoa Essentials

This article describes the basic concepts and features of the garbage collection technology that are essential for a Cocoa developer to understand. It does not provide a complete treatment of the subject—you are expected to read the other articles in this document to gain a deeper understanding. In particular, you should also read "Implementing a finalize Method" (page 31).

Basic Concepts

When you use the Cocoa garbage collection technology, it manages your application's memory for you. All Cocoa objects are garbage collected. There is no need to explicitly manage objects' retain counts to ensure that they remain "live" or that the memory they take up is reclaimed when they are no longer used. For example, with garbage collection enabled the following method (although inefficient!) does not result in any memory leaks:

```
- (NSString *)fullName {
    NSMutableString *mString = [[NSMutableString alloc] init];
    if ([self firstName] != nil)
        [mString appendString:[self firstName]];
    if (([self firstName] != nil) && ([self lastName] != nil))
        [mString appendString:@" "];
    if ([self lastName] != nil)
        [mString appendString:[self lastName]];
    return [mString copy];
}
```

How the Garbage Collector Works

The garbage collector's goal is to form a set of **reachable** objects that constitute the "valid" objects in your application, and then to discard any others. When a collection is initiated, the collector initializes the set with all well-known **root** objects. The collector then recursively follows **strong** references from these objects to other objects, and adds these to the set. At the end of the process, all objects that are *not* reachable through a chain of strong references to objects in the root set are designated as "garbage." At the end of the collection sequence, the unreachable objects are finalized and immediately afterwards the memory they occupy is recovered.

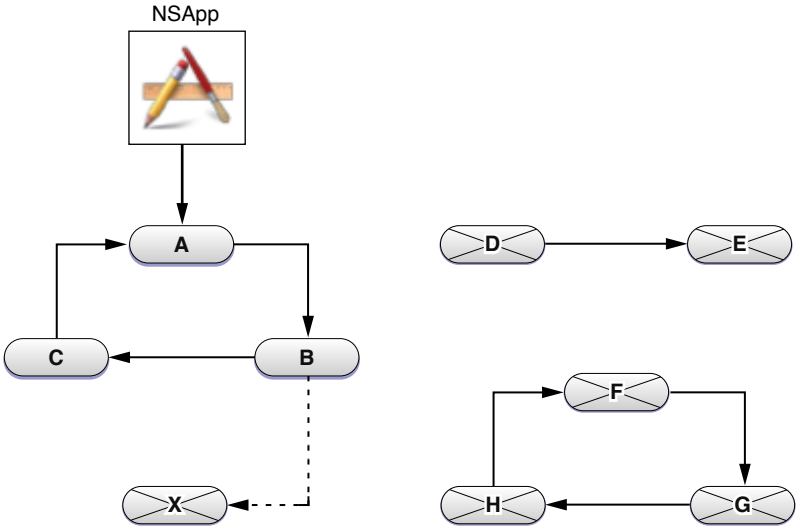
Root Set and Reference Types

The initial root set of objects is comprised of global variables, stack variables, and objects with external references (for more details about globals, see "Global Object Pointers" (page 27)). These objects are never considered as garbage. The root set is comprised of all objects reachable from root objects and all possible references found by examining the call stacks of every Cocoa thread.

Important: Note that the optimizing compiler can greatly reduce the lifespan of variables on the stack, reusing stack slots as it determines that slot is no longer used by the code. This can result in objects being collected more quickly than you might expect—see for example "Interior Pointers" (page 27).

Conversely, there may be situations in which you inadvertently leave references beyond the top of the stack which the collector might so consider live and hence delay collection of the corresponding objects. For more details, see `objc_clear_stack` in "Language Support" (page 48).

As implied earlier, there are two types of reference between objects—strong and weak. A strong reference is visible to the collector, a weak reference is not. A non-root object is only live if it can be reached via strong references from a root object. An important corollary is that simply because you have a strong reference to an object does not mean that that object will survive garbage collection, as illustrated in the following figure.



There is a strong reference from a global object (the shared `NSApplication` instance) to object A, which in turn has a strong reference to B, which has a strong reference to C. All of these objects are therefore valid. There is a weak reference from B to X, therefore X will be treated as garbage.

There is a strong reference from D to E, but since neither has a strong reference from a root object, both are treated as garbage. As an extension of the latter case, objects F, G, and H illustrate a retain cycle. In reference-counted applications this may be a problem (see Object Ownership and Disposal); in a garbage collected application, since none of these objects has a strong reference from a root object all are treated as garbage and all are properly reclaimed.

All references to objects (`id`, `NSObject *`, and so on) are considered strong by default. Objects have strong behavior, but so can other memory blocks and Core Foundation-style objects. You can create a weak reference using the keyword `__weak`, or by adding objects to a collection configured to use weak references (such as `NSHashTable` and `NSMapTable`).

Enabling Garbage Collection

Garbage collection is an optional feature; you need to set an appropriate flag for the compiler to mark code as being GC capable. The compiler will then use garbage collector write-barrier assignment primitives within the Objective-C runtime. An application marked GC capable will be started by the runtime with garbage collection enabled.

There are three possible compiler settings:

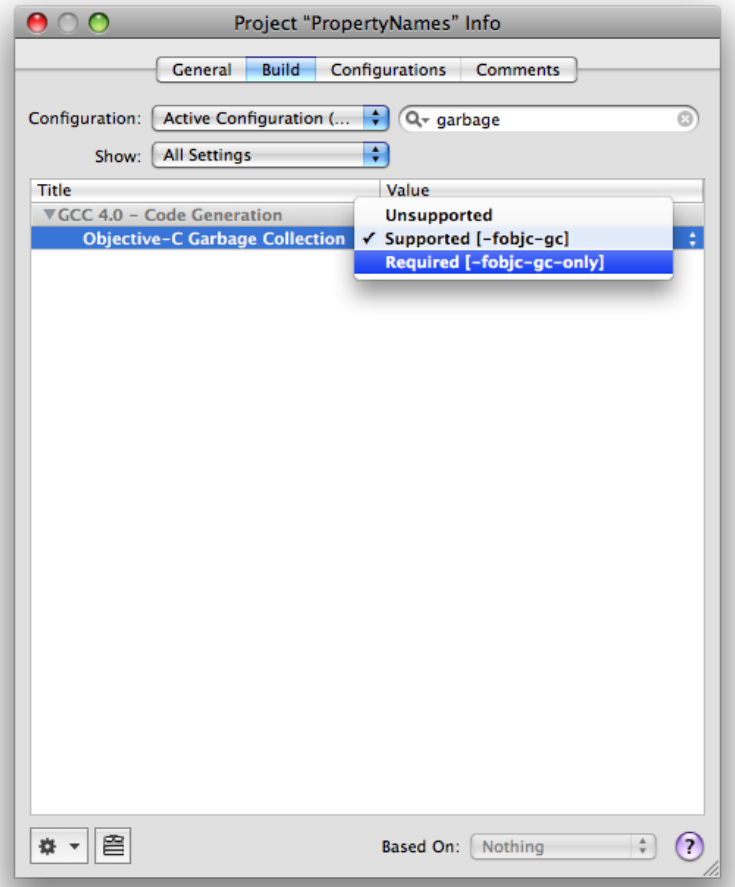
- No flag. This means that GC is not supported.
- `-fobjc-gc-only` This means that only GC logic is present.

  Code compiled as GC Required is presumed to not use traditional Cocoa retain/release methods and may not be loaded into an application that is not running with garbage collection enabled.
- `-fobjc-gc` This means that both GC and retain/release logic is present.

  Code compiled as GC Supported is presumed to also contain traditional retain/release method logic and can be loaded into any application.

You can choose an option most easily by selecting the appropriate build setting in Xcode, as illustrated in Figure 1 (page 14).

Figure 1 Xcode code generation build settings for garbage collection



Foundation Tools

In a Cocoa desktop application, the garbage collector is automatically started and run for you. If you are writing a Foundation tool, you need to start the collector thread manually using the function `objc_startCollectorThread`:

```
int main (int argc, const char * argv[]) {
    objc_startCollectorThread();
    // your code
    return 0;
}
```

You may want to occasionally clear the stack using `objc_clear_stack()` to ensure that nothing is falsely rooted on the stack. You should typically do this when the stack is as shallow as possible—for example, at the top of a processing loop.

You can also use `objc_collect(OBJC_COLLECT_IF_NEEDED)` to provide a hint to the collector that collection might be appropriate—for example, after you finish using a large number of temporary objects.

Design Patterns to Use, and to Avoid

Don't try to optimize details in advance.

Finalizing objects

In a garbage-collected application, you should ideally ensure that any external resources held by an object (such as open file descriptors) are closed prior to an object's destruction. If you do need to perform some operations just before an object is reclaimed, you should do so in a `finalize` method. For more details, see "Implementing a finalize Method" (page 31). Note that you should never invoke `finalize` directly (except to invoke super's implementation in the `finalize` method itself).

Don't manage scarce resources with object lifecycle

If an object holds on to a scarce resource, such as a file descriptor, you should indicate that the resource is no longer required using an invalidation method. You should *not* wait until the object is collected and release the resource in `finalize`. For more details, again see "Implementing a finalize Method" (page 31).

Nib files

Since the collector follows strong references from root objects, and treats as garbage all objects that cannot be reached from a root object, you must ensure that there are strong references to all top-level objects in a nib file (including for example, stand-alone controllers)—otherwise they will be collected. You can create a strong reference simply by adding an outlet to the File's Owner and connecting it to a top-level object. (In practice this is rarely likely to be an issue.)

Triggering garbage collection

In a standard application, Cocoa automatically hints at a suitable point in the event cycle that collection may be appropriate. The collector then initiates collection if memory load exceeds a threshold. Typically this should be sufficient to provide good performance. Sometimes, however, you may provide a hint to the collector that collection may be warranted—for example after a loop in which you create a large number of temporary objects. You can do this using the `NSGarbageCollector` method `collectIfNeeded`.

```
// Create temporary objects
NSGarbageCollector *collector = [NSGarbageCollector defaultCollector];
[collector collectIfNeeded];
```

Threading

Garbage collection is performed on its own thread—a thread is explicitly registered with the collector if it calls `NSThread`'s `currentThread` method (or if it uses an autorelease pool). There is no other explicit API for registering a `pthread` with the collector.

Prune caches

The collector scans memory to find reachable objects, so by definition keeps the working set hot. You should therefore make sure you get rid of objects you don't need.

Avoid allocating large numbers of short-lived objects

Object allocation is no less expensive an operation in a garbage collected environment than in a reference-counted environment. You should avoid creating large numbers of (typically short-lived) objects.

Compile GC-Only

In general, you should not try to design your application to be dual-mode (that is, to support both garbage collection and reference-counted environments). The exception is if you are developing frameworks and you expect clients to operate in either mode.

C++

In general, C++ code should remain unchanged: you can assume memory allocated from standard `malloc` zone. If you need to ensure the longevity of Objective-C objects, you should use `CFRetain` instead of `retain`.

Adopting Garbage Collection

Garbage collection provides trade-offs that you need to consider when choosing whether to adopt the technology.

Potentially, any application that uses a runloop may use garbage collection, however there are issues you should consider when deciding whether it is appropriate for your application. Garbage collection provides several advantages when compared with reference counting; there are also, though, some disadvantages. The benefits tend to be greater if the application is threaded and has a reasonably large working set; they tend to be less if the latency of memory recovery is important. Moreover, reference-counted and garbage collected applications use a number of different idioms and patterns.

For information relating to garbage collection in the current release of Mac OS X, see Garbage Collection Release Notesâ aLÂMOSÂ XÂ v10.5.0.

Note: The process of migrating a large project that uses reference counting can be difficult and error-prone—some patterns that work correctly with manual memory management will be incorrect after translation. In general, it is recommended that you use garbage collection only in new projects. If you already have a well-architected, well-understood application that uses reference counting, there should be little reason to migrate to GC.

Advantages and Disadvantages

Garbage collection offers some significant advantages over a reference-counted environment:

- Most obviously, it typically simplifies the task of managing memory in your application and obviates most of the memory-related problems that occur, such as retain cycles.
- It reduces the amount of code you have to write and maintain, and may make some aspects of development easier—for example, zeroing weak references facilitate use of objects that may disappear.
- It usually makes it easier to write multi-threaded code: you do not have to use locks to ensure the atomicity of accessor methods and you do not have to deal with per-thread autorelease pools. (Note that although garbage collection simplifies some aspects of multi-threaded programming, it does not automatically make your application thread-safe. For more about thread-safe application development, see *Threading Programming Guide*.)

Garbage collection does though have some disadvantages:

- The application's working set may be larger.
- Performance may not be as good as if you hand-optimize memory management (for more details, see "Performance" (page 18)).
- A common design pattern whereby resources are tied to the lifetime of objects does not work effectively under GC.

- You must ensure that for any object you want to be long-lived you maintain a chain of strong references to it from a root object, or resort to reference counting for that object.

Performance

The performance characteristics of an application that uses garbage collection are different from those of an application that uses reference-counting. In some areas, a garbage-collected application may have better performance, for example:

- Multi-threaded applications may perform better with garbage collection because of better thread support;
- Accessor methods are much more efficient (you can implement them using simple assignment with no locks);
- Your application is unlikely to have leaks or stale references.

In other areas, however, performance may be worse:

- Allocation may be a significant consideration if your application allocates large numbers of (possibly short-lived) objects.
- The working set may be larger—in particular, the overall heap can grow larger due to allocation outpacing collection.
- The collector scans heap memory to find reachable objects, so by definition keeps the working set hot. This may be a significant consideration, particularly if your application uses a large cache.
- The collector runs in a secondary thread. As such, a GC-enabled application will in almost all cases consume more CPU cycles than a reference-counted application.

When analyzing the performance of a garbage-collected application, you typically need to take a longer-term approach than with a reference-counted application. When assessing its memory footprint, it may be appropriate to measure after the application has been running for several minutes since the memory footprint may be greater shortly after launch. The profiling tools you can use include `heap`, `gdb` flags, and the Instruments application.

Architecture

Garbage collection simplifies memory management and makes it easier to ensure thread and exception safety. It also avoids common problems such as retain cycles, and simplifies some code patterns (such as accessor methods in Cocoa). Together these make applications more robust.

Motivation and Design Goals

Garbage collection systems were first developed around 1960 and have undergone much research and refinement since then. Most garbage collection systems restrict direct access to memory pointers. This has the benefit that you never have to be concerned about memory errors—either leaks due to cyclic data structures or due to the use of a dangling pointer. The Objective-C language, however, has no such restrictions on pointer use. Although a few garbage collection systems have been developed for use with the C language, their assumptions and performance make them unsuitable for use with Cocoa objects. Cocoa therefore uses a custom non-copying conservative garbage collection system that in normal use brings safety and a simplified programming model.

Restricted pointer access-languages allow for fully-automatic garbage collection. If you program purely in objects, then garbage collection in Cocoa can also be fully automatic. Beyond programming purely in objects, however, the collector also provides access to a new collection-based memory allocation system. Core Foundation objects are also garbage collected, but you must follow specific rules to allocate and dispose of them properly. In order to understand how you can take advantage of these features, you need to understand some of the architectural details described in this document.

The immediate benefits of garbage collection can be highlighted using a simple class definition and implementation. The Widget class is declared as follows:

```
@interface Widget : NSObject
{
@private
    Widget *nextWidget;
}
- (Widget *)nextWidget;
- (void)setNextWidget:(Widget *)aWidget;
@end
```

Listing 1 (page 19) illustrates a full-featured, thread-safe, traditional Cocoa implementation of the Widget class.

Listing 1 Full-featured implementation of the Widget class

```
@implementation Widget
- (Widget *)nextWidget
{
    @synchronized(self)
    {
        return [[nextWidget retain] autorelease];
```

```
    }
}

- (void)setNextWidget:(Widget *)aWidget
{
    @synchronized(self)
    {
        if (nextWidget != aWidget)
        {
            [nextWidget release];
            nextWidget = [aWidget retain];
         }
    }
}
@end
```

There are many other permutations that trade increased speed for less safety (see Basic Accessor Methods).

If you do not implement memory management correctly, your application will suffer from memory leaks that bloat its memory footprint, or even worse, from dangling pointers which lead to crashes. Retain cycles, or circular references, can cause significant problems in traditional Cocoa programming (see, for example, Object Ownership and Disposal). Consider the following trivial example.

```
Widget *widget1 = [[Widget alloc] init];
Widget *widget2 = [[Widget alloc] init];
[widget1 setNextWidget:widget2];
[widget2 setNextWidget:widget1];
```

If you use manual memory management and the accessor methods described earlier, this sets up a retain cycle between the two widgets and is likely to lead to a memory leak.

If you use a garbage collector, the implementation of the Widget class is much simpler.

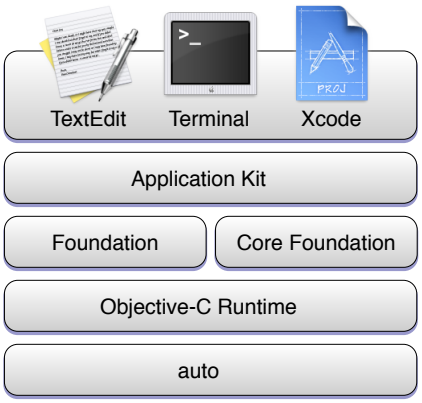
```
@implementation Widget
- (Widget *)nextWidget
{
    return nextWidget;
}

- (void)setNextWidget:(Widget *)aWidget
{
    nextWidget = aWidget;
}
@end
```

Retain cycles are not a problem if you use garbage collection: as soon as both objects become unreachable, they are marked for deletion.

High-Level Architecture

The garbage collector is implemented as a reusable library (called “auto”). The Objective-C runtime is a client of the library.

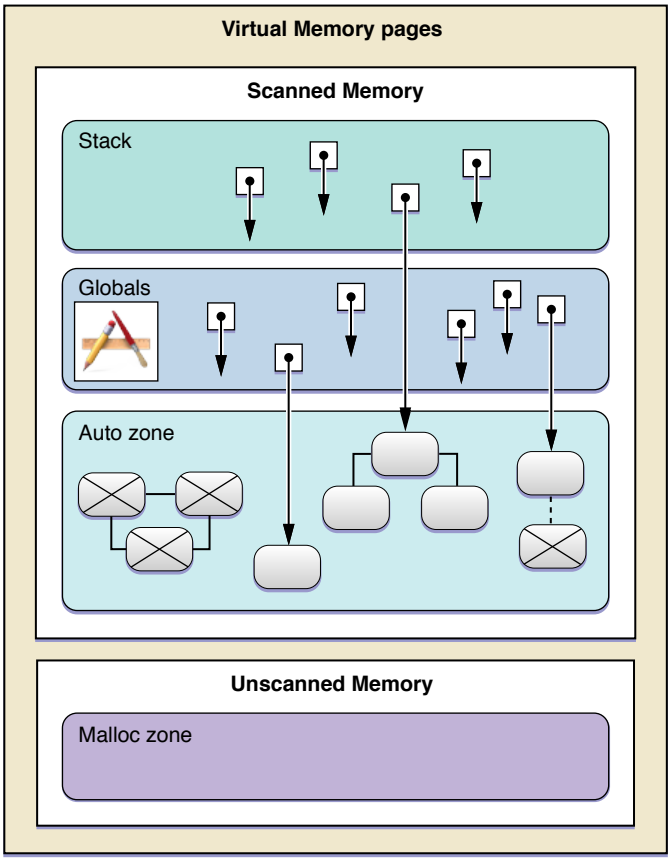


The collector does not scan all areas of memory (see Figure 1 (page 22)). The stack and global variables are always scanned; the malloc zone is never scanned. The collector provides a special area of memory known as the auto zone from which all garbage-collected blocks of memory are dispensed. You can use the collector to allocate blocks of memory in the auto zone—these blocks are then managed by the collector.

How the Garbage Collector Works

The mechanism of garbage collection is fairly simple to describe although the implementation is more complicated. The garbage collector's goal is to form a set of **reachable** objects that constitute the "valid" objects in your application. When a collection is initiated, the collector initializes the set with all known **root** objects such as stack-allocated and global variables (for example, the `NSApplication` shared instance). The collector then recursively follows **strong** references from these objects to other objects, and adds these to the set. All objects that are *not* reachable through a chain of strong references to objects in the root set are designated as "garbage". At the end of the collection sequence, the garbage objects are finalized and immediately afterwards the memory they occupy is recovered.

Figure 1 Scanned and unscanned memory



There are several points of note regarding the collector:

- The collector is conservative—it never compacts the heap by moving blocks of memory and updating pointers. Once allocated, an object always stays at its original memory location.
- The collector is both request and demand driven. The Cocoa implementation makes requests at appropriate times. You can also programmatically request consideration of a garbage collection cycle, and if a memory threshold has been exceeded a collection is run automatically.
- The collector runs on its own thread in the application. At no time are all threads stopped for a collection cycle, and each thread is stopped for as short a time as is possible. It is possible for threads requesting collector actions to block during a critical section on the collector thread's part. Only threads that have directly or indirectly performed an `[NSThread self]` operation are subject to garbage collection.
- The collector is generational (see "Write Barriers" (page 23))—most collections are very fast and recover significant amounts of recently-allocated memory, but not all possible memory. Full collections are also fast and do collect all possible memory, but are run less frequently, at times unlikely to impact user event processing, and may be aborted in the presence of new user events.

Closed vs. Open Systems

Most garbage collection systems are “closed”—that is, the language, compiler, and runtime collaborate to be able to identify the location of every pointer reference to a collectable block of memory. This allows such collectors to reallocate and copy blocks of memory and update each and every referring pointer to reflect the new address. The movement has the beneficial effect of compacting memory and eliminating memory wastage due to fragmentation.

In contrast to closed collection systems, “open” systems allow pointers to garbage collected blocks to reside anywhere, and in particular where pointers reside in stack frames as local variables. Such garbage collectors are deemed "conservative." Their design point is often that since programmers can spread pointers to any and all kinds of memory, then all memory must be scanned to determine unreachable (garbage) blocks. This leads to frequent long collection times to minimize memory use. Memory collection is instead often delayed, leading to large memory use which, if it induces paging, can lead to very long pauses. As a result, conservative garbage collection schemes are not widely used.

Cocoa's garbage collector strikes a balance between being “closed” and “open” by knowing exactly where pointers to scanned blocks are wherever it can, by easily tracking "external" references, and being "conservative" only where it must. By tracking the allocation age of blocks, and using write barriers, the Cocoa collector also implements partial (“incremental” or “generational”) collections which scan an even smaller amount of the heap. This eliminates the need for the collector to have to scan all of memory seeking global references and provides a significant performance advantage over traditional conservative collectors.

Write Barriers

In most applications, objects are typically short-lived—they are created on a temporary basis, consulted, and never used again. Cocoa's Garbage Collector is **generational**—it divides allocated memory into "generations" and prioritizes recovery of memory from the newest generations. This means that the memory from short-lived objects can often be reclaimed quickly and easily.

In order to recover these objects, the compiler introduces what is known as a write-barrier whenever it detects that an object pointer is stored (“assigned”) into another object, or more completely, whenever a pointer to a garbage collected block of memory is stored into either another garbage collected block (or into global memory).

What Does a Write-Barrier Do?

Within the collector, memory is split into several generations—old and newer. The write-barrier simply marks a "clump" of objects when a "newer" object is stored somewhere within an older. The number of "clumps" of older generation objects that get marked is usually very low. When an incremental garbage collection is requested, the stack and the objects within marked clumps are examined recursively for "newer" objects that have been attached and are now reachable. These "newer" objects are then marked "older" (promoted). All unreachable "newer" objects are reclaimed after any necessary finalization.

A generational collection does not discover any older generation objects that are no longer reachable and so, over time, the oldest generation needs to be examined with a "full" collection. In principle there can be many generations—a generational collection in the midst of work with a lot of temporary objects will promote the temporary objects to an older generation where they could be recovered without resorting to a full collection. The Cocoa collector runs with 2 to 8 generations.

Write-Barrier Implementation

Consider the following example.

```
static id LastLink;
@interface Link2 : NSObject {
    id theLink;
}
- link;
- (void)setLink:newLink;
@end

@implementation Link2
- link {
    return theLink;
}
- (void)setLink: newLink
{
    theLink = newLink;
    LastLink = newLink;
}
@end
```

Behind the scenes the compiler calls an intrinsic helper function to deal with the assignment and when garbage collection is enabled the helper function calls into the collector to note the store of a pointer. Effectively the two assignments within `setLink:` are rewritten by the compiler to be:

```
objc_assignIvar(newLink, self, offsetof(theLink));
objc_assignGlobal(newlink, &LastLink);
```

These helper functions are almost without cost when not running with garbage collection—there is only a two instruction penalty. At runtime, if garbage collection is enabled these routines are rewritten at startup to include the write-barrier logic.

Using Garbage Collection

This article describes some of the design patterns and features you can take advantage of when you use garbage collection, and some of subtleties you need to be aware of.

Cycles

A problem when using manually reference counting is that it is possible to create retain cycles. If two objects retain each other, and you do not have a reference to either, then they will remain valid for the lifetime of your application—constituting a memory leak (see, for example, Object Ownership and Disposal).

With garbage collection, retain cycles are not a problem. Since the collector traces strong reference from root objects, even if two objects have strong references to each other they can be collected if neither has a reference from a root object.

Weak and Zeroing Weak References

Sometimes you need a reference to an object but do not want to form a strong relationship to that object to prevent its being collected if it has no other references. For example, a notification center should not form strong relationships to registered observers, otherwise it artificially prolongs the lifetime of those objects indefinitely. You can specify a weak reference—one that the collector does not follow—using the keyword `__weak`.

`NSMapTable`, `NSHashTable`, and `NSPointerArray` provide collection objects that have the option of maintaining zeroing weak references to their elements. If an element is collected, the reference from the collection object is simply removed.

Managing Opaque Pointers

There are several Cocoa methods and Core Foundation functions that have as one parameter an opaque pointer (`void *`). In a garbage collected environment, the general policy is that the lifetime of any object passed as a `void *` should be either managed by the callbacks or known to be safe.

For example, in Cocoa, there are a several "asynchronous" methods that take a delegate object, a selector, and a context and send the selector to the delegate object at some later point passing the context as an argument. These APIs typically declare the context as a `void *` and represent it as such in their internal state. A common example of this kind of code flow is seen with sheets, especially sheets that are created by a temporary controller object as illustrated in the following code fragment:

```
@implementation MySheetController
```

```
- (IBAction)showDoSomethingSheetAction:(id)action
{
    id contextObject = /* ... */;
    // code omitted

    // point A
    [NSApp beginSheet:sheetWindow
           modalForWindow:window
           modalDelegate:delegate
           didEndSelector:@selector(sheetDidEnd:returnCode:contextInfo:);
           contextInfo:(void *)contextObject];
}
@end

@implementation MySheetControllerDelegate
- (void)sheetDidEnd:(NSWindow *)sheet returnCode:(int)returnCode contextInfo:(void
 *)contextInfo
{
    // point B
    id contextObject = (id)contextInfo;

    [contextObject doSomething];
    // ...
}
@end
```

The problem is that in between points A and B, a garbage collection can occur and—if there are no strong references to it from a root object—the context object can be collected. (This example is somewhat simplified, but in a complex application it's a situation that can happen when the only strong reference to the object passed via the context parameter is on the stack—which for a sheet will be unwound all the way to the main run loop.)

The solution is to use a `CFRetain`/`CFRelease` pair as the value is put into/taken out of the `void *` parameter. This ensures that the object that will be used as context won't be collected until after it's no longer used (see "Memory Management Semantics" (page 39)).

```
@implementation MySheetController
- (IBAction)showDoSomethingSheetAction:(id)action
{
    id contextObject = /* ... */;
    // code omitted

    // point A
    CFRetain(contextObject);

    [NSApp beginSheet:sheetWindow
           modalForWindow:window
           modalDelegate:delegate
           didEndSelector:@selector(sheetDidEnd:returnCode:contextInfo:);
           contextInfo:(void *)contextObject];
}
@end

@implementation MySheetControllerDelegate
- (void)sheetDidEnd:(NSWindow *)sheet returnCode:(int)returnCode contextInfo:(void
 *)contextInfo
{
    // point B
```

```
    id contextObject = (id)contextInfo;
    // code omitted
    [contextObject doSomething];
    CFRelease(contextObject);
}
@end
```

Global Object Pointers

Typically, the garbage collector treats global object pointers as root objects and so does not consider them candidates for collection (see "Root Set and Reference Types" (page 11)). Globals of Objective-C objects or other `__strong` pointer variables, and function-level static variables, are written to with a write-barrier. Note that although this is true for Objective-C or Objective-C++, writing to globals from C or C++ is not supported. Weak globals have the same restriction; in addition, however, you cannot read from them in C or C++.

You can check whether a write-barrier is being used with the `-Wassign-intercept` compiler flag—see "Compiler Flag" (page 29).

Limitations on Mac OS X v10.5: You may pass addresses of strong globals or statics into routines expecting pointers to object pointers (such as `id*` or `NSError**`) *only* if they have first been assigned to directly, rather than through a pointer dereference. You should never take the address of a weak global, static or instance variable, as assigning or reading through that pointer will bypass the weak barriers and expose your programs to race conditions.

Assigning a value to a global by reference does not work correctly, as illustrated in the following example. Given these global declarations:

```
static id globalId1;
static id globalId2;
```

if in a function you make an assignment to a global by reference:

```
id localObject = ... ;
id *localPointerToGlobal = (someTest() ? &globalId1 : &globalId2);
*localPointerToGlobal = localObject;
```

then the wrong write barrier is used in the assignment of `localObject`. You can work around this by using `objc_assign_global`.

Interior Pointers

The compiler can reuse stack slots it determines are no longer used (see "Root Set and Reference Types" (page 11)). This can mean that objects are collected more quickly than you might expect—when a local variable is removed from the stack and hence the corresponding object not considered rooted. This has implications for situations in which you access data held by a local variable after the last direct reference to that variable. To illustrate, consider the following example:

```
NSData *myData = [someObject getMyData];
const uint8_t *bytes = [myData bytes];
NSUInteger offset = 0, length = [myData length];
```

```
while (offset < length) {
  // if you never reference myData again, bytes is a dangling pointer.
}
```

Suppose that after you send `myData` the `length` message, you do not reference it again directly. The compiler may reuse the stack slot for `myData`. `myData` may then become eligible for collection (see "Root Set and Reference Types" (page 11)); if it is collected, then `bytes` becomes invalid.

You can ensure that the data object remains valid until you've finished using it by sending it a message, as shown in the following version:

```
NSData *myData = [someObject getMyData];
[myData retain];
const uint8_t *bytes = [myData bytes];
NSUInteger offset = 0, length = [myData length];

while (offset < length) {
  // bytes remains valid until next message sent to myData
}
[myData release];
```

Alternatively, in this particular case you can retrieve data from the object by sending it messages, as in this variant:

```
NSData *myData = [someObject getMyData];
const uint8_t *bytes = [myData bytes];
NSUInteger currentAddress = 0, finalAddress = [myData length];

while (currentAddress < finalAddress) {

    NSRange range = NSMakeRange (currentAddress, bytes[currentAddress]);
    if (range.length > finalAddress || currentAddress > finalAddress -
range.length) {
        // This is an overflow
        break;
    }

    NSData *newData = [NSData dataWithBytesNoCopy:(void *)&bytes[currentAddress]
                                           length:length freeWhenDone:NO];
    currentAddress += length;
    // use the data from newData...
}
```

This ensures that `myData` remains on the stack until after you have finished processing the data it contains.

C Structs

If you assign an Objective-C object into a raw C `struct`, the compiler will treat the reference as strong, even if you cast an `id` directly to a `void *`:

```
MyStruct *myStruct;
id object;
// assign the variables appropriately...
myStruct->anObject = object;
myStruct->anObject = (void *)object;
```

Both the assignments to `aField` result in strong references.

Compiler Flag

You can use the `-Wassign-intercept` compiler flag to find out when write-barriers are being used in your code. When you set the flag, the compiler logs a message when it offloads an assignment statement to a helper function.

Typically you use the flag to find situations where a write-barrier is *not* actually used. A missing write-barrier can cause various problems—primarily when memory is being collected before you expect it to be. This is of particular interest when you're using pointers to garbage-collected memory—especially pointers to pointers—such as where a left-hand-side cast discards the strong knowledge, as illustrated in the following example:

```
__strong CFDictionaryRef x;  // struct or ivar declaration
*(CFMutableDictionaryRef)&x = CFDictionaryCreateMutable(...);
```

Implementing a finalize Method

This article describes how to correctly and efficiently implement a `finalize` method.

Design Patterns

Object finalization occurs at most once during the lifetime of an object—when it is collected. When more than one object is finalized, the order in which they are sent a `finalize` message is indeterminate, even if there are references between them. If you send messages between objects when they are being finalized, you must take extreme care to avoid anomalous behavior. To ease correctness concerns alone, it is best not to attempt any work in a finalizer. Moreover, however, time spent in object finalization incurs application overhead. Your design goal should therefore be to not have a finalizer at all. If you must use a finalizer, you should keep it as short as possible, and reference as few other objects as possible in its implementation.

Efficiency

Memory recovery time is typically *not* the best time to reclaim resources or do clean-up work (such as releasing instance variables and closing resources). Your `finalize` code is part of the garbage collector's critical path, and so should be kept to a minimum if not eliminated entirely. You should implement invalidation code that is distinct from your deallocation or finalization code and invoke it when appropriate.

To make your `finalize` method as efficient as possible, you should typically *not* do any of the following:

- Disconnect object graphs
- Set instance variables to `nil`
- For view classes, remove `self` from the existing view hierarchy
- Remove `self` as an observer of a notification center (in a garbage collected environment, notification centers use zeroing weak references).

You should typically use `NSMakeCollectable()` on Core Foundation objects rather than relying on `CFRelease()` in `finalize`—this way collectable Core Foundation objects are actually collected sooner. (Collectable objects are collected with the source object whereas released objects are simply marked as being eligible for collection—these must wait for the next collection cycle to be collected.)

Messaging Other Objects

No objects are deallocated until all finalizers are complete (otherwise, no finalizer could use any other object anywhere, including objects like `NSString` that don't have a finalizer) so you can access already-finalized objects—but only in other finalizers. Within a `finalize` method, therefore, you should reference as few other objects as possible. You can't necessarily know what other objects might have a reference to your

instance, or whether they might message your instance from their finalizer, you must therefore code defensively to try to keep your instance as fully functional as is possible to support messages it might receive after finalization. Similarly, since you don't know in what order objects will be finalized, it may be that objects you message during a `finalize` method have themselves already been cleared.

For example, some objects use collection objects (arrays, dictionaries, or sets) to hold other related objects. Sometimes during finalization the collection is accessed and messages sent to each and every contained object. If the collection itself had been finalized and had discharged its objects, the algorithm would fail on that account alone. Similarly, if any of the objects in the collection can no longer respond correctly to the requested message after it is finalized, the algorithm again will fail.

Avoiding Resurrection

Some Cocoa objects make assumptions about how many references are kept about themselves and where, for example by implementing the `release` method to trap the transition to a known value (typically of `0`) and then distributing cleanup work among their holders. In a garbage-collected environment, this pattern can lead to "resurrection" of an object—that is, it becomes valid again after having been finalized.

Resurrection occurs when a `finalize` method stores `self` in a non-garbage object. The resurrected object becomes a zombie. It logs all messages that are sent to it, but it is otherwise useless. It is eventually deallocated when it becomes garbage again (when its container is collected). You should consider resurrection to be a programming error.

The following example illustrates a trivial, albeit unlikely, case:

```
- (void)finalize
{
    [NSArray arrayWithObject:self];
}
```

Managing an External Resource

The following example illustrates what happens if an object must manage an external resource—in this case, a Logger object is given a file descriptor to use for writing logging messages. File descriptors are not inexhaustible, and so the object provides a `close` method to relinquish the resource. In an ideal scenario, you should have closed the file descriptor before the `finalize` method is called. If, however—as is implied in this example—you have a shared or singleton object, it may not be possible to actively manage the object's resources, and you will have to rely on `finalize` to clean up. To ensure that the file descriptor is not kept beyond the object's lifetime, therefore, the `close` method is invoked in the `finalize` method.

```
@interface Logger : NSObject
{
    int fileDescriptor;
}
- initWithFileDescriptor:(int)aFileDescriptor;
- (void)close;
- (void)log:(NSString *)message;
@end

@implementation Logger
- initWithFileDescriptor:(int)aFileDescriptor
```

```
{
    fileDescriptor = aFileDescriptor;
    return self;
}

- (void)close
{
    if (fileDescriptor != -1) close(fileDescriptor);
    fileDescriptor = -1;
}

- (void)finalize
{
    [self close];
    [super finalize];
}

- (void)log:(NSString *)message
{
    // Implementation continues ...
}
@end
```

The runtime invokes the `finalize` method after it determines that a logger object can no longer be reached. The message is sent once and it is an error for a finalizing object to have a new reference created to it in a reachable object. In other words, the object may not be revived (resurrected) once found to be unreachable.

A problem emerges even in this simple example. What would happen if a Logger object were created to track some other "larger" object, for example a window or a drawer or a network connection? This larger object might offer a logging API that enabled notations to be delivered to the file descriptor to mark progress. It might be natural to then have in this larger object one last message in its finalizer:

```
- (void)finalize
{
    [logger log:@"saying goodbye!"];
    [logger close];
    [super finalize];
}
```

Unfortunately the results would not always match your expectation, because the final message would sometimes appear and sometimes not. This is because the larger object and the logger object would both be found to be garbage in the same collection cycle and both would be put on the finalizer list in some order, and that order would require that the logger be after the larger object in order for the file descriptor resource to still be open.

Inapplicable Patterns

The following sections discuss design patterns that are applicable in Cocoa applications that use reference counting but which do not translate well to a garbage collected environment.

Reference counting

If you use garbage collection, the methods that are used to implement the manual reference counting system (`retain`, `release`, `dealloc`, `autorelease`, and `retainCount`) have no effect—the Objective-C messenger short circuits their dispatch. As a result, overriding `release` and `dealloc` is not supported when garbage collection is enabled—this makes obsolete some object caching patterns.

Note, however, that `CFRetain` and `CFRelease` do still have an effect in Core Foundation objects. See also "Adopting Garbage Collection" (page 17).

dealloc

When you use "classic" memory management, you typically implement a `dealloc` method that performs "clean-up" operations such as releasing instance variables, unregistering the object from a notification center, and closing resources. In a garbage-collected application, the analog of the `dealloc` method is `finalize`.

In a garbage-collected application, there is obviously no need to release instance variables, however you should ideally ensure that other resources are closed prior to an object's destruction. For more details, see "Implementing a finalize Method" (page 31).

Although there are conceptual similarities between `dealloc` and `finalize`, there are some constraints on the implementation of `finalize` that do not apply to `dealloc`. In particular, you must ensure that there are no ordering issues.

Occasionally, within a completely captive subgraph, significant work is done in `dealloc` methods as they do recursive releases and subsequent deallocations. Many applications that use reference counting make use of the deterministic ordering of object deallocation. If one object A retains another object B, A can guarantee that during its `dealloc` method the B is valid (object B's `dealloc` method has not been called) and so send B messages and otherwise interact with it.

If you use garbage collection, it is possible for A and B to become invalid at the same time. Moreover, there is no ordering of the invocation of objects' `finalize` methods. If object A has a strong reference to object B, and object A and object B are both reclaimed during a given collection cycle, then there is no guarantee that object A's `finalize` method will be invoked first. Object A cannot therefore make any assumptions about the state of object B in its `finalize` method. Or, conversely, object B must be prepared to be messaged after its `finalize` method is invoked.

Since `finalize` messages may be sent in any order, existing code that relies on side effects during `dealloc` methods will need to introduce new methods to achieve a similar graph walking effect.

Enumerations

If you use weak collections, the count of the collection may change during an iteration loop. This will obviously lead to problems if you iterate over the contents of the collection directly using a `for` loop. On the other hand, enumeration objects can cause resurrection of the collection or its objects if all are found to be garbage at the same time—this is particularly likely to occur if you use a pattern where you have a collection of helper objects and on finalization they perform cleanup work (see "Avoiding Resurrection" (page 32)).

To avoid these problems, you should use the `NSFastEnumeration` protocol (see Fast Enumeration) to iterate over the contents of a collection.

Resource wrapper objects

A common pattern is to associate an object with an external resource—for example, a file descriptor—that needs "management" or other state that the object coordinates, often across several threads. The typical implementation is to use a non-retaining CFDictionary coupled with a global lock at the lookup and deallocation stages. This pattern does not work when you use garbage collection because there is a timing window during finalization where the object is no longer reachable from a root, yet is still in the dictionary and can be resurrected.

The solution is to use an `NSMapTable` object. A map table can hold keys, values, or both weakly, and when the objects are discovered unreachable the table is immediately cleared of such entries before any finalization is performed. This prevents resurrection of the object being finalized. For resources created and destroyed within the application, such as file descriptors, this is an adequate solution.

Leaked but not lost objects

Cocoa used to have several classes of object (fonts and images) where a global table of strong keys held weak value references to the objects. The object would remove itself from the global table on `dealloc`. But it would also be the case that there would be some universally known objects that never went away, and the pattern was to allocate these at startup using `[[alloc ] init]` and simply place them in the weak table. The reference count for these objects would never decrease and so they would live indefinitely. Under garbage collection, in the absence of a strong reference these universal objects are collected. The solution is to use `[[NSGarbageCollector defaultCollector] disableCollectorForPointer:object]` on these objects before placing them in the weak table.

Delegate references

If you do not use garbage collection, references to delegates are typically “weak” (in that the delegate is not retained). This is to avoid retain cycle problems. With garbage collection, retain cycles do not pose a problem, so there is no need to declare references to delegates as `__weak`.

Memory zones

You cannot allocate objects in separate zones—all Cocoa objects must be allocated in a single managed heap. If your application is running in garbage collection mode, the zone parameter in `NSAllocateObject` is ignored. With garbage collection enabled, `[NSObject allocWithZone:zone]` calls `NSAllocateObject(cls, extra, zone)`, which in turn calls `objc_allocate_object(cls, extra)`.

You can allocate memory such that it is scanned using `NSAllocateCollectable` or `NSReallocateCollectable`.

Using Core Foundation with Garbage Collection

Sometimes you want to integrate Core Foundation objects into your application. If your application uses garbage collection, you must then ensure that you manage the memory for these objects correctly.

Core Foundation provides C-based opaque types for a number of data-types—including strings and dates and numbers and collections—that have counterparts in Cocoa Objective-C classes (for example, CFString corresponds to `NSString`). There are also Core Foundation opaque objects that don't have a direct Objective-C counterpart, yet also respond to basic Objective-C messages (such as `hash` and `isEqual:`). These opaque data types can be treated by Cocoa as objects—for example, they can be stored in collections. Since these objects are nearly indistinguishable from those created directly in Objective-C, they are also allocated and collected by the garbage collector, although they do require some special handling at time of creation.

Allocation

The collection system supports multiple memory zones. When you create a Core Foundation object, you specify the zone using the allocator parameter. In a garbage collected environment, the standard default Core Foundation allocator (which normally points to the default malloc zone) is aimed at one that uses the garbage collector system—so by default all Core Foundation objects are allocated by the collector. The following list summarizes the behavior of the allocators in a garbage collected environment:

- `NULL`, `kCFAllocatorDefault`, and `kCFAllocatorSystemDefault` specify allocation from the garbage collection zone.

  By default, all Core Foundation objects are allocated in the garbage collection zone.
- `kCFAllocatorMallocZone` specifies allocation from default `malloc` zone.
- `kCFAllocatorMalloc` specifies allocation explicitly with `malloc()` and deallocation with `free()`.

Memory Management Semantics

Because you can use Core Foundation objects in applications that use garbage collection or reference counting, the Core Foundation memory management functions `CFRetain()` and `CFRelease()` are required to interoperate correctly in either environment. As a policy, they function in the same way in both—they respectively increment and decrement the reference counts of Core Foundation objects.

In a garbage collected environment, the `CFRetain` and `CFRelease` implementations are redirected to also use the garbage collectors reference counting mechanism. *The collector does not collect any object with a non-zero count* (or any object reachable from such an object—Core Foundation objects with a retain count greater than zero act as root objects). Within C based code, therefore, `CFRetain` and `CFRelease` still perform the same logical functions that they always do—it's just that the memory source and the location of the reference count bits is different.

Note: You can take advantage of this feature if you have to store a reference into non-scanned memory and cannot (and should not) guarantee that a valid reference exists elsewhere. This is similar to creating a JNI Global Reference to hold onto a Java object from C code. See also, though, `disableCollectorForPointer:`.

By default, therefore, in a garbage-collected environment you manage Core Foundation objects exactly as you would in a reference-counted environment (as described in *Memory Management Programming Guide for Core Foundation* > Ownership Policy). If you create or copy a Core Foundation object, you must subsequently release it when you're finished with it. If you want to keep hold of a Core Foundation object, you must retain it and again subsequently release it when you're finished with it.

The difference between the garbage-collected environment and reference-counted environment is in the timing of the object's deallocation. In a reference counted environment, when the object's retain count drops to 0 it is deallocated immediately; in a garbage-collected environment, what happens when a Core Foundation object's retain count transitions from 1 to 0 depends on where it resides in memory:

- If the object is in the malloc zone, it is deallocated immediately.
- If the object is in the garbage collected zone, the last `CFRelease()` does not immediately free the object, it simply makes it eligible to be reclaimed by the collector when it is discovered to be unreachable—that is, once all strong references to it are gone. Thus as long as the object is still referenced from an object-type instance variable (that hasn't been marked as`__weak`), a register, the stack, or a global variable, it will not be collected.

This behavioral difference gives you some additional flexibility in a garbage collected environment. In a non-garbage-collected application you call `CFRelease()` only when you want to relinquish ownership; in a garbage-collected application you may call `CFRelease()` immediately after allocation and the object will be collected when appropriate. Better still, though, you can use `CFMakeCollectable` instead of `CFRelease`. `CFMakeCollectable` calls `CFRelease`, but has two supplementary features: first, it halts the program if the object wasn't allocated in the scanned zone; second, it's a no-op in a reference counted environment. (In addition, it more clearly signals your intent.) For example:

```
CFStringRef myCFString = CFMakeCollectable(CFStringCreate...(...));
```

You can also use `NSMakeCollectable`. This is exactly the same as `CFMakeCollectable` except that it returns an `id`—you can use this to avoid the need for casting, as illustrated in the following example:

```
id myNSString = NSMakeCollectable(CFStringCreate...(...));
```

You could imagine the implementation of CFMakeCollectable as being similar to this:

```
id CFMakeCollectable(CFTypeRef object)
{
    if (object != NULL)
    {
        CFAllocatorRef allocator = CFGetAllocator(object);
        if ((allocator != kCFAllocatorDefault) && (allocator !=
kCFAllocatorSystemDefault))
        {
            // register an error
        }
        CFRelease([(id)object retain]);
    }
    return object;
}
```

Similarly, you could define a hypothetical `MakeUncollectable()` function as follows:

```
id MakeUncollectable(id object)
{
    [CFRetain(object) release];
    return object;
}
```

This makes a currently collectable object uncollectable by giving it a retain count of 1.

There are three important corollaries here:

1. A single `CFMakeCollectable` (and hence `NSMakeCollectable`) balances a single `CFRetain`. For example, absent any additional memory management code, the following code fragment will result in `myCFString` "leaking":

   ```
   CFStringRef myCFString = CFMakeCollectable(CFStringCreate...(...));
   CFRetain(myCFString);
   ```

   You must balance the `CFRetain` with a further `CFMakeCollectable`.

2. Because `CFMakeCollectable` is a no-op in a reference counted environment, if you use it with mixed mode code you do need to use `CFRelease` when running without garbage collection.

   ```
   CFStringRef myCFString = CFMakeCollectable(CFStringCreate...(...));
   // do interesting things with myCFString...
   if ([NSGarbageCollector defaultCollector] == NULL) CFRelease(myCFString);
   ```

3. It is important to appreciate the asymmetry between Core Foundation and Cocoa—where `retain`, `release`, and `autorelease` are no-ops. If, for example, you have balanced a CFCreate… with `release` or `autorelease`, you will leak the object in a garbage collected environment:

   ```
   NSString *myString = (NSString *)CFStringCreate...(...);
   // do interesting things with myString...
   [myString release]; // leaked in a garbage collected environment
   ```

   Conversely, using `CFRelease` to release an object you have previously retained using `retain` will result in a reference count underflow error.

Core Foundation Variables

Instance variables

The garbage collector can only track a reference if it knows that it should be treated as an object. If you declare a Core Foundation structure as an instance variable, the compiler regards it only as an opaque structure pointer, not as an object. Assignments will not therefore by default generate the write-barriers required by the collector, the compiler needs some explicit information—this is also true for Core Foundation variables declared globally.

To indicate that a Core Foundation structure should be treated as a collectable object, you use the `__strong` keyword. This denotes that scanned memory references are to be stored into a given variable and that write-barriers should be issued.

```
@interface MyClass
__strong CFDateRef myDate;
@end

@implementation MyClass

- (id)init
{
    if (self = [super init])
    {
        myDate = CFMakeCollectable(CFDateCreate(NULL, 0));
    }
    return self;
}

/*
There is no need for a finalize method here
*/
@end
```

If you want to see when write barriers are generated, you can ask the compiler to emit a warning at every point it issues a write-barrier by using the `Wassign-intercept` flag.

Local Variables

If you allocate a Core Foundation object locally, you can use `CFRetain` and `CFRelease` just as you would in a non-garbage collected application, for example:

```
- (void)doSomethingInterestingWithALocalCFDate
{
    CFDateRef epoch = CFDateCreate(NULL, 0);
    // ...
    CFRelease(epoch);
}
```

If you return the value, however, to ensure that the returned value is eligible for collection you must balance the Create with `NSMakeCollectable` (or `CFMakeCollectable`) as illustrated in the following example:

```
- (id)anInterestingDate
{
    CFDateRef epoch = CFDateCreate(NULL, 0);
    // ...
    return NSMakeCollectable(epoch);
}
```

If you are writing mixed-mode code (code that has to run in both a garbage-collected and reference-counted environments), you can use `NSMakeCollectable` (or `CFMakeCollectable`) to bring Core Foundation objects into the `NSObject` world as shown in this example (remember that `CFMakeCollectable` is a no-op in a reference-counted environment and `autorelease` is a no-op in a garbage collected environment):

```
- (NSString *)languageForString:(NSString *)string
{
    CFStringTokenizerRef tokenizer;
    // create and configure the tokenizer...
   CFStringRef language = CFStringTokenizerCopyCurrentTokenAttribute(tokenizer,
 kCFStringTokenizerAttributeLanguage);
```

```
    CFRelease(tokenizer);
    return [NSMakeCollectable(language) autorelease];
}
```

Core Foundation Collection Semantics

Collections (such as arrays and dictionaries) allocated in the scanned zone use strong references instead of reference counting (this is important for good garbage collection performance).

```
__strong CFMutableArrayRef myList;
```

Core Foundation collection objects such as dictionaries have different properties than their Objective-C Cocoa counterparts. In particular, they allow for non-retained entries which need not be objects but may be other pointers or even values of pointer size. This allows you, for example, to use integers as keys in a dictionary object. To accomplish this you pass `NULL` callbacks at collection object creation. This has the effect of just copying the pointer sized value into the collection object with no additional processing.

When the values are in fact objects they are stored as non-retained (weak) pointers, and if those objects are somehow reclaimed, what is stored becomes dangling references. Although unsafe, this practice is correctly supported when running under GC. Both the standard retaining as well as the non-retaining, weak (`NULL`) callbacks are supported correctly.

Garbage Collection API

This article summarizes the classes, methods, and functions associated with garbage collection.

Foundation

Foundation provides several classes to help support design patterns associated with garbage collection, and the behavior of several methods in existing classes is changed when running under garbage collection.

In prior releases of Mac OS X, `NSHashTable` and `NSMapTable` were opaque structure pointers that were configured and used with C function callout structures. C functions were used to access `void *` elements. In Mac OS X v10.5, these structures have been minimally converted to objects and exactly preserve the behaviors of prior releases. In addition, however, the `NSHashTable` and `NSMapTable` objects feature Objective-C-based API patterned after `NSSet` and `NSDictionary` respectively. Both classes offer the ability to configure the tables using zero-ing weak pointer memory when running under garbage collection (GC), as well as the ability to copy elements when input, or alternatively to have the objects be treated using pointer identity and hashing. The `void *` C functions, as well as the new methods, work on both kinds of table.

NSObject

`NSObject` adds the `finalize` method; other methods listed below are ignored completely or have changed semantics when used in a garbage collected environment.

+allocWithZone:(NSZone *)zone
: The *`zone`* argument is ignored.

- (id)autorelease
: This method is a no-op.

-(void)dealloc
: This method is a no-op.

-(void)finalize
: Conceptually similar to the traditional `dealloc`—for more details, see "Implementing a finalize Method" (page 31).

- (oneway void)release
: This method is a no-op.

- (id)retain
: This method is a no-op.

- (NSUInteger)retainCount
: The return value is undefined.

NSAutoreleasePool

`NSAutoreleasePool` adds the `drain` method.

-(void)drain

Triggers garbage collection if memory allocated since last collection is greater than the current threshold. (This method ultimately calls `objc_collect_if_needed()`.)

NSGarbageCollector

`NSGarbageCollector` provides an object-oriented abstraction of the garbage collector. You use `defaultCollector` to return the collector (this returns `nil` in a reference-counted environment).

You can use `disableCollectorForPointer:` to ensure that memory at a given address will not be scanned—for example, to create new root objects. You balance this with `enableCollectorForPointer:`, which makes collectable memory that was previously marked as uncollectible.

Collection Classes

`NSHashTable` is a new collection class like `NSMutableSet` but which (amongst other features) provides the ability to create weak references to its contents.

`NSMapTable` is a new collection class like `NSMutableDictionary` but which (amongst other features) provides the ability to create weak references to its contents.

`NSPointerArray` is a new collection class like `NSArray` but it can also hold `NULL` values, which can be inserted or extracted (and contribute to the object's count). Also unlike traditional arrays, you can set the count of the array directly. Under Garbage Collection and using a zeroing weak memory configuration, `NULL` values appear when elements are collected. A pointer array uses an instance of `NSPointerFunctions` to define callout functions appropriate for managing a pointer reference held somewhere else.

NSValue

`NSValue` has a method to wrap a non-retained object, `valueWithNonretainedObject:`.

+(id)valueWithNonRetainedObject:(id)anObject

Creates a new `NSValue` object containing a weak reference to `anObject`. If `anObject` is garbage collected, the reference is set to `nil`.

NSThread

`NSThread` provides additional functionality for `currentThread`.

`currentThread`	Returns the thread object representing the current thread of execution.

Foundation Functions

Various functions have been added.

void *NSAllocateCollectable(NSUInteger size, NSUInteger options)
: Allocates *size* bytes of memory using the given option.

id NSAllocateObject(Class aClass, NSUInteger extraBytes, NSZone *zone);
: The zone parameter is ignored by `NSAllocateObject` in GC mode.

id NSMakeCollectable(CFTypeRef cf)
: This function is a wrapper for `CFMakeCollectable` (see "Core Foundation Functions" (page 47)), but its return type is `id`, avoiding the need to cast if you assign the value to a Cocoa object.

  This function may be useful when returning Core Foundation objects in code that must support both garbage-collected and non-garbage-collected environments, as illustrated in the following example.

```
- (NSString *)description
{
    CFStringRef myCFString = CFStringCreate...(...);
    return [NSMakeCollectable(myCFString) autorelease];
}
```

Core Foundation

The behavior of several functions is different under garbage collection. The Core Foundation collection types (such as CFSet, CFMutableSet, CFDictionary, and CFArray) correctly support the standard "retaining" callbacks under GC in a way that allows cycles to be recovered, unlike non-GC behavior. Note also that `NULL` callbacks will weakly reference objects, but are not done with zeroing memory—you still need to remove objects from the collection. If you need zeroing weak object behavior, use `NSHashTable` or `NSMapTable` instead.

CFMutableArray

Changed semantics when creating with `NULL` arguments.

CFArrayCreateMutable(NULL, 0, NULL)
: References contents weakly, does *not* zero. You must remove objects from the array.

CFMutableDictionary

Changed semantics when creating with `NULL` arguments.

CFDictionaryCreateMutable(NULL, 0, NULL, NULL)
: References contents weakly, does *not* zero. You must remove objects from the dictionary.

Core Foundation Functions

New and changed functions.

CFTypeRef CFMakeCollectable(CFTypeRef anObject)

Checks that anObject is a Core Foundation object allocated in the scanned memory zone and, in a garbage collected environment, releases it. This function is a no-op in a reference-counted environment.

void CFRelease(CFTypeRef anObject)

Decrements the retain count for `anObject`. If `anObject` was allocated in a garbage collected zone, then if its retain count is reduced to zero it is not actually deallocated until next collection. If `anObject` was allocated in a malloc zone, then if its retain count is reduced to zero it is deallocated immediately. Thus for GC objects, `CFRelease()` no longer has immediate side-effects.

Language Support

Features and functions.

`__strong`

Specifies a reference that is visible to (followed by) the garbage collector (see "How the Garbage Collector Works" (page 11)).

`__strong` modifies an instance variable or struct field declaration to inform the compiler to unconditionally issue a write-barrier to write to memory. `__strong` is implicitly part of any declaration of an Objective-C object reference type. You must use it explicitly if you need to use Core Foundation types, `void *`, or other non-object references (`__strong` modifies *pointer* assignments, not scalar assignments).

`__strong` essentially modifies all levels of indirection of a pointer to use write-barriers, except when the final indirection produces a non-pointer l-value. When you declare a variable, you can put `__strong` on either side of the `*`; in the following example, all the variable declarations are equivalent:

```
@interface MyClass {
    __strong int *ptr1;
    int * __strong ptr2;
    int __strong * ptr3;
}
```

`__weak`

Specifies a reference that is *not* visible to (followed by) the garbage collector (see "How the Garbage Collector Works" (page 11)).

`__weak` informs the compiler to use a zeroing weak reference for variables. All writes are done using a weak write-barrier, and all reads use a weak read-barrier. This allows you to reference a variable without preventing the variable from being garbage collected.

Weak references are set to zero (`nil`) if the destination is collected. If an object is collected, any weak instance variables are cleared *before* being finalized. Thus, in a `finalize` method, you can guarantee that any `__weak` instance variables have been set to `nil`.

Runtime

The runtime provides a number of functions to support different aspects of garbage collection, and an environment variable you can use to check whether GC is on or off for a process.

`objc_allocate_object(cls, extra)`

Allocates a new object.

`id objc_msgSend(id theReceiver, SEL theSelector, ...)`

Ignores these selectors: `retain`, `release`, `autorelease`, `retainCount`, `dealloc`. This is faster than messaging `nil`.

`void objc_collect_if_needed(int options)`

Triggers garbage collection if memory allocated since last collection is greater than the current threshold. Pass `OBJC_GENERATIONAL` to run generational collection.

This function must only be called from the main thread.

`void objc_clear_stack(unsigned long options)`

This function may be useful if you write your own event loop type mechanisms or code not using run loops—you need to clear the stack or risk unnecessarily extending the lifetime of objects.

Any uninitialized local variables will hold whatever values happen to be on the stack from previous function calls. Those values may be pointers to old objects which, while you don't consider the objects to still be live, the garbage collector will still see a reference to them on the stack and not collect them because of it. For example, if a function has a local variable which is a `char` array of `MAX_PATH` length, and you read a line and fill it with only a dozen characters, everything past the end of the dozen characters is left over from previous function calls and may be interpreted by the collector as pointers to objects.

OBJC_PRINT_GC

When debugging, you can perform a runtime check by setting the environment variable `OBJC_PRINT_GC=YES`. This prints the GC state of each Objective-C image, and whether GC is on or off for the process.

Document Revision History

This table describes the changes to *Garbage Collection Programming Guide*.

Date	Notes
2008-11-19	Added note that Foundation programs need to use objc_startCollectorThread().
2008-10-15	Clarified behavior of CFMakeCollectable.
2008-03-11	Corrected typographical errors.
2007-12-11	Added an article to discuss integrating Core Foundation and garbage collection.
2007-10-31	Corrected minor typographical errors.
2007-07-12	Corrected minor typographical errors.
Leopard WWDC	New document that describes the garbage collection system for Cocoa.